



信息学院软件工程系 **《实用操作系统》实验报告**

**题 目** ELF文件格式与虚拟内存实验设计

**任课老师** 李贵林 **组员姓名** 周彦妮 任宇 向泽旭 左泽宇 侯妤欣 曾予岑

**提交时间 2023** 年 **12** 月 **06** 日

**ELF文件格式与虚拟内存实验设计**

1. **实验目的**
   * 根据ELF文件格式，修改ELF文件，实现预期效果。
   * 观察ELF文件中各个段如何被映射到虚拟内存中。
2. **实验环境**
   * 操作系统：
     + - 主机：Windows 10
       - 虚拟机：Ubuntu 18.04
   * 开发板：IMAX6ULL MIN
   * 文件传输工具：FileZilla
   * 终端工具：MobaXterm
3. **实验思路**

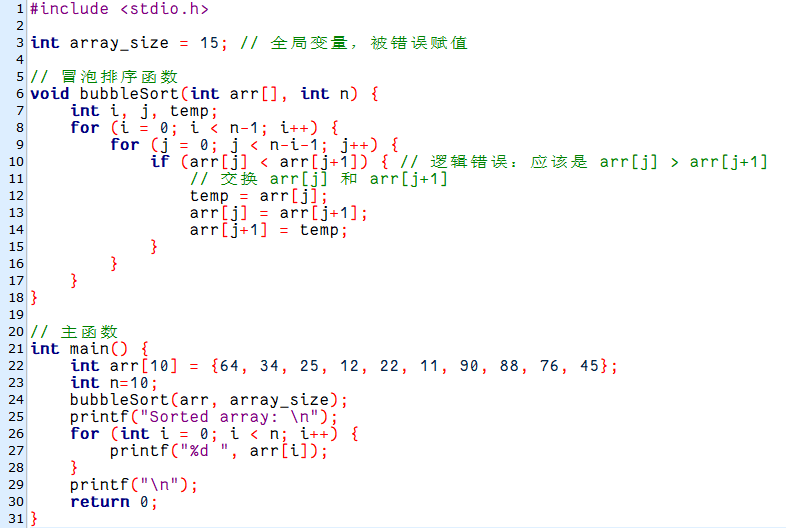
* 对于实验目的一，通过学习ELF文件相关知识，我们可以知道：
* ELF（Executable and Linkable Format）文件是一种广泛使用的文件格式，用于定义可执行文件、目标代码、共享库和核心转储，它是标准的二进制文件格式。它主要有以下几个组成部分：
* 头部：描述了整个文件的布局和属性，包括魔数、文件类型（如可执行文件、共享对象）、机器类型（如 x86、ARM）、入口点地址（程序开始执行的地方）等。
* 程序头表：描述了程序运行时需要的各个段，这些段在加载时被映射到虚拟内存中，同时也包括段的类型、偏移、虚拟地址、物理地址、文件大小、内存大小、权限和对齐。
* 节头表：描述了文件中的节（如代码、数据、符号表、重定位信息）。主要用于链接和调试。
* 节：包含程序的实际数据，如代码（.text）、数据（.data、.bss）、符号表（.symtab）、字符串表（.strtab）、重定位信息等。
* 段：由程序头表描述，用于程序的加载。一个段可以包含多个节，它描述了如何将节映射到内存中。

基于这些基本了解，本次实验将对ELF文件中的text(代码)段和data(数据)段进行修改，我们首先需要找到其在ELF文件中的位置，然后进行我们期望的修改以实现预期的结果。

* 对于实验目的二，我们则是通过阅读LiteOS-A中实现ELF加载和执行的源代码，找出并修改其中执行内存映射的函数，增加打印功能，以便我们观察ELF文件中指出的地址与实际映射的虚拟内存地址间的关系。

1. **实验步骤**
2. **修改ELF文件**
3. 这部分实验研究如何针对ELF可执行文件实现代码注入，这是一个简单的示例，但却可以帮助理解ELF文件的格式。

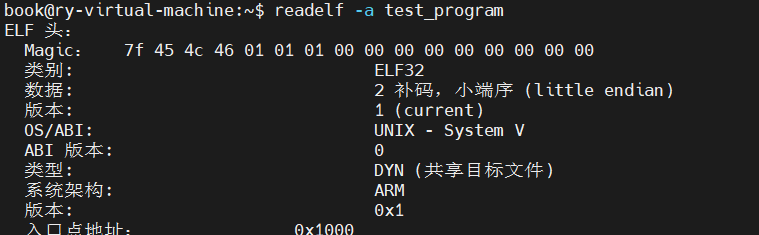
首先编写一段代码，然后将其编译成可执行的ELF文件：



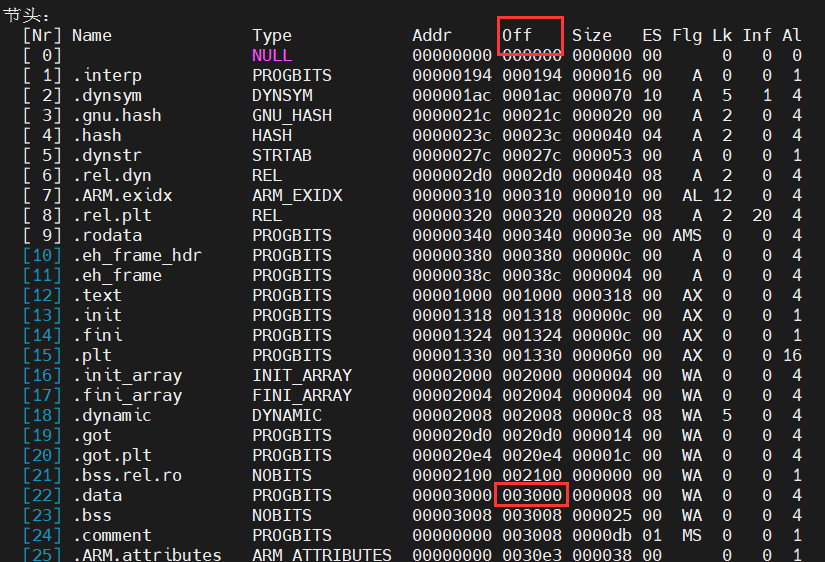


这个程序有一个明显的错误，那就是全局变量array\_size被错误赋值，这会导致越界问题。

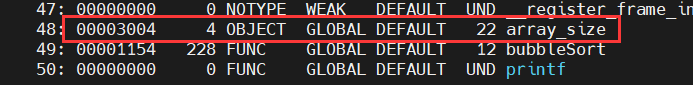
1. 全局变量被存储在data段中，为了解决这个问题，我们首先需要找出data段以及array\_size变量在ELF文件中的位置，使用readelf文件检查文件(文件名为test\_program)：



查看节头部分，发现data段相对文件起始位置的偏移是0x3000：

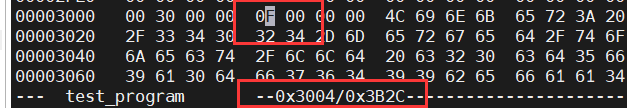


接着查看符号表部分，找到array\_size符号，可以发现其位置为0x3004：

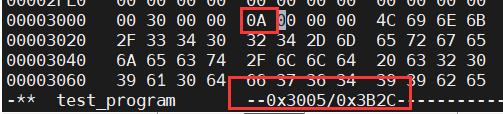


1. 使用HexEdit修改ELF文件(hexedit test\_program)，并将程序加载到liteos.bin文件中，接着在开发板上运行观察结果：

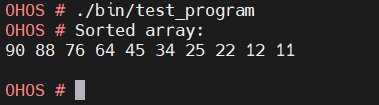
可以发现，test\_program文件中0x3004的位置确实存储了array\_size的值（15），我们将其修改为10。



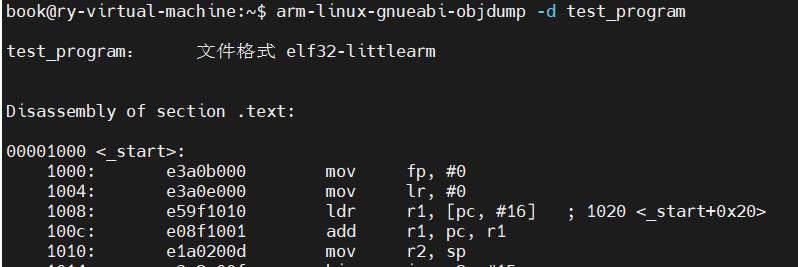
可以发现，test\_program文件中0x3004的位置确实存储了array\_size的值（15），我们将其修改为10：

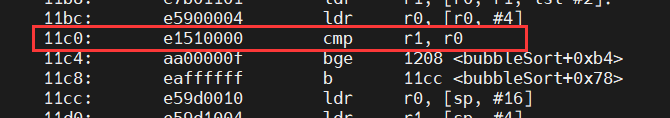


可见程序正确运行：



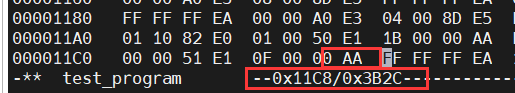
1. 现在test\_program实现的是降序排序，如果我们想实现升序排序且不更改并编译源文件，我们该怎么做呢？一种简单的方法是修改ELF文件中代码段的机器指令，首先使用objdump命令反汇编test\_program，找出实现BubbleSort函数的部分，接着确定待修改指令的位置：

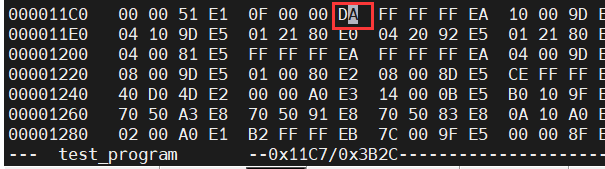


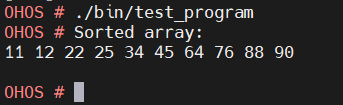


在这里，cmp r1, r0指令比较r1和r0寄存器的值，这两个寄存器分别加载了 arr[j] 和 arr[j+1] 的值。紧接着的 bge 指令是一个条件跳转，它基于比较的结果来决定是否跳转到指定的地址。在当前的降序排序实现中，如果 arr[j] 大于等于 arr[j+1]，则跳转继续下一个循环迭代。要改为升序排序，我们需要在 arr[j] 小于 arr[j+1] 时进行交换。因 此，需要更改 bge 指令为 blt 指令，它会在 arr[j] 小于 arr[j+1] 时跳转。机器码中，bge 指令的操作码是 AA，而 jl 指令的操作码是 DA。 同时可以得到的信息是，这个命令位于test\_program的0x11C4位置。

1. 使用HexEdit修改ELF文件，并将程序加载到liteos.bin文件中，接着在开发板上运行观察结果：

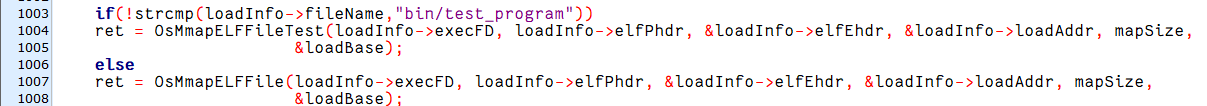




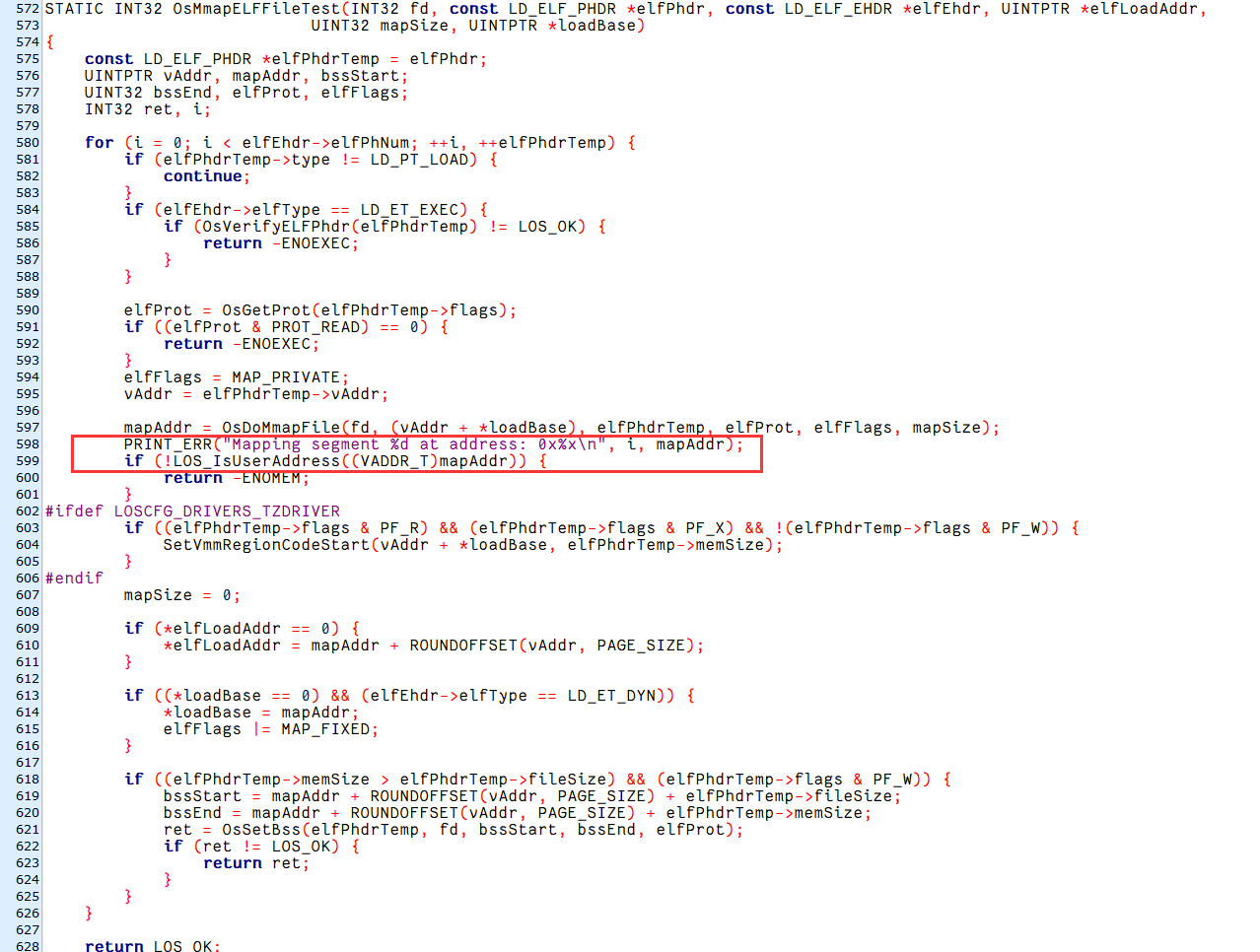


可以发现程序成功改为升序排序。

1. **观察ELF如何映射到虚拟内存**
2. 修改los\_load\_elf.c文件，为其增加一个测试用的函数，当加载特定名称的ELF文件时打印段地址（此处为test\_program），修改OsLoadELFSegment函数，增加下图所示代码：

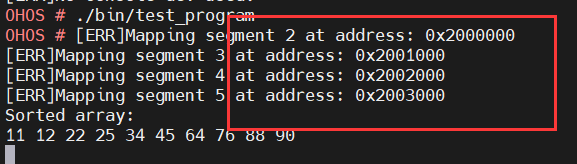


1. 编写测试函数，这里的测试函数参照OsMmapELFFile函数设计，只不过在每次循环中打印段地址：

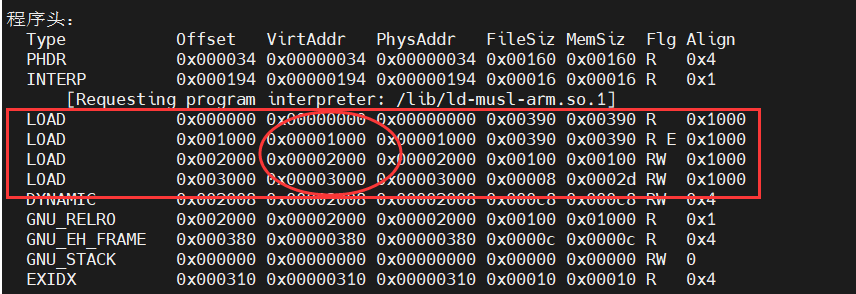


为了方便在用户态下观察打印信息，这里使用PRINT\_ERR输出。函数会遍历 ELF 文件的程序头数组并过滤非加载段。函数仅处理类型为 LD\_PT\_LOAD 的段，这些是需要被加载到内存的段。接着会使用 OsDoMmapFile 函数将段映射到虚拟内存。映射地址存储在 mapAddr 变量中。如果返回正确，对于每个映射的段，打印映射的段号和地址，以便于调试和验证。

1. 编译内核，并加载到开发板中运行：



1. 观察输出结果，并于ELF文件中指出的虚拟地址作比较：

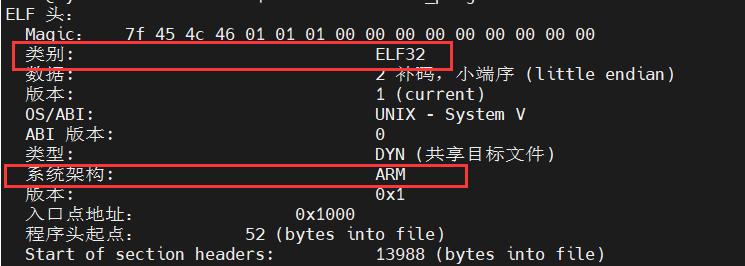


观察ELF的程序头，我们可以发现0x00000000、0x00001000、0x00002000、0x00003000 这些地址被映射到了0x2000000、0x2001000、0x2002000、0x2003000。也就是说在加载时，加载器将ELF文件中指出的这些地址映射到了实际的虚拟内存地址上。这是因为操作系统为每个进程提供了一个独立的虚拟地址空间。ELF 文件中指定的虚拟地址是相对于这个地址空间的起始地址。同时还要注意一种情况，那就是操作系统加载器在将程序加载到内存时有可能进行地址重定位。这是一种常见的情况，尤其是在支持地址空间布局随机化（ASLR）的系统中。

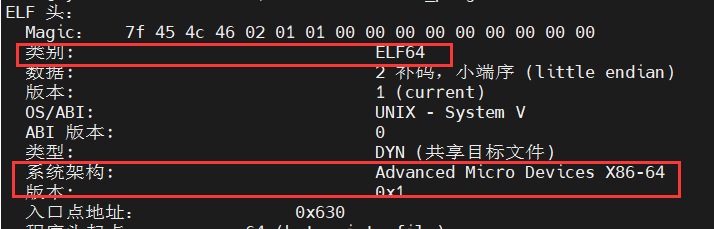
1. 补充：在Ubuntu18.04中的ELF文件与LiteOs-A中的ELF文件有何不同？

首先最明显的区别就是两者格式不同，但这是由于编译导致的：

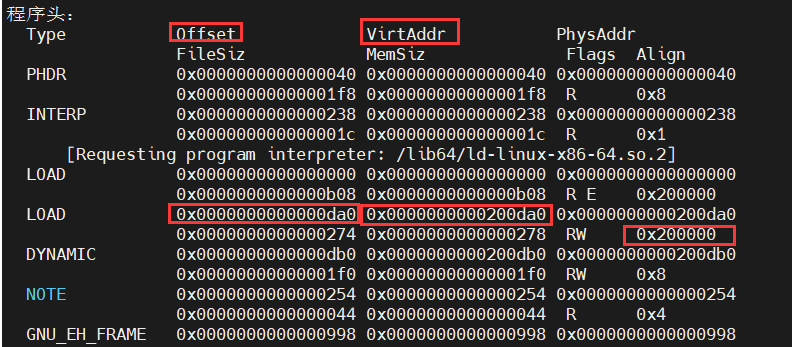
LiteOS-A：



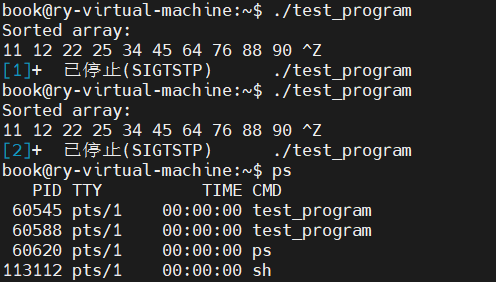
Ubuntu18.04:



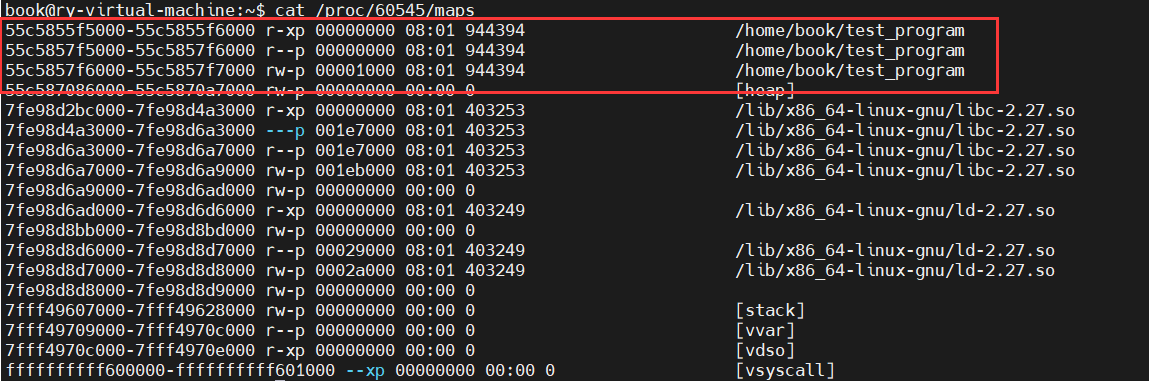
我们可以观察Ubuntu18.04平台上能运行的ELF文件头，会发现其也为各个段指定了映射到哪个虚拟地址：

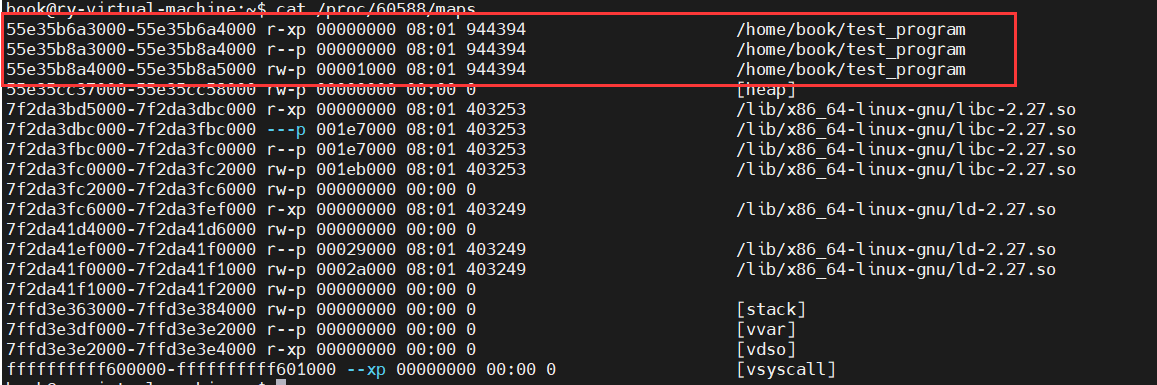


其中offset是指相对于ELF文件的偏移量，而VirtAddr则是指定的虚拟内存地址，我们运行这个程序并观察其内存分布：



执行两个test\_program进程，并查看其/proc/pid/maps：





可以发现尽管执行的是同一个ELF程序，但是装入到虚拟内存时却是装入了不同的地址。选择一个进行观察：

55c5855f5000-55c5855f6000 r-xp /home/book/test\_program

//这个段标记为 r-xp，即可读、可执行，不可写。它对应于包含程序代码的 LOAD 段。在程序头表中，第一个 LOAD 段（位于文件偏移 0x00000000）是可执行的（标记为 RE），与这个段匹配。

55c5857f6000-55c5857f7000 rw-p /home/book/test\_program

//对应ELF文件头中的数据段，0x201000+55c5855f5000=55c5857f6000

1. **实验总结**

本次实验的目标是深入理解ELF（Executable and Linkable Format）文件格式，并观察ELF文件如何被映射到虚拟内存中。我们首先学习了ELF文件的基本组成部分，包括文件头部、程序头表、节头表和各个节。特别地，我们关注了ELF文件中的text（代码）段和data（数据）段，以及这些段如何映射到内存中。

实验的第一部分涉及对ELF文件进行修改。我们通过修改data段中的变量值和text段中的机器指令，实现了对程序行为的控制。例如，我们成功修改了全局变量的值，以及改变了程序的排序算法。这不仅展示了ELF文件的灵活性，也加深了我们对程序编译和链接过程的理解。

第二部分的实验重点在于观察ELF文件映射到虚拟内存的过程。通过修改LiteOS-A内核的源代码和增加打印功能，我们能够观察到不同段的映射地址。这一过程帮助我们理解了操作系统如何将程序文件中的地址映射到进程的地址空间，以及地址重定位的机制。

总体而言，这次实验不仅加深了我们对ELF文件格式的理解，还让我们对程序如何被操作系统加载到内存中有了更加直观的认识。通过实际操作和观察，我们更加深刻地理解了程序运行时的内存管理和映射机制。此外，实验过程中的挑战也提升了我们的问题解决能力和团队合作技巧。

1. **遇到的问题及如何解决**
2. 在反汇编test\_program文件时，发现使用objdump并不能够处理arm架构的文件，解决方案是：

安装 ARM 架构的交叉编译工具链。对于 ARM，通常的包是 binutils-arm-linux-gnueabi。使用 ARM 版本的 objdump 来反汇编ELF 文件：

***arm-linux-gnueabi-objdump -d test\_program***

1. 在修改los\_load\_elf.c文件时发现，如果只是在内核代码中增加PRINTK函数是无法在用户态下输出信息的，解决方案是：

使用PRINT\_ERR宏进行打印，这样就可以在用户态下输出信息且不会影响程序的正常运行。

1. **参考文献**
2. [**深入浅出ELF - 知乎 (zhihu.com)**](https://zhuanlan.zhihu.com/p/254882216)
3. [**一个elf程序实现代码注入的实例-腾讯云开发者社区-腾讯云 (tencent.com)**](https://cloud.tencent.com/developer/article/1821538)
4. [**kernel\_liteos\_a: LiteOS kernel for embedded devices with rich resources | 适用于资源较丰富嵌入式设备的LiteOS内核 (gitee.com)**](https://gitee.com/openharmony/kernel_liteos_a)